Étape 13 : Termites et ennemis.

Jamila Sam & Jean-Cédric Chappelier, 2018

Version : 1.0

But

Compléter la modélisation des termites et permettre aux animaux de reconnaître leurs ennemis.

Description

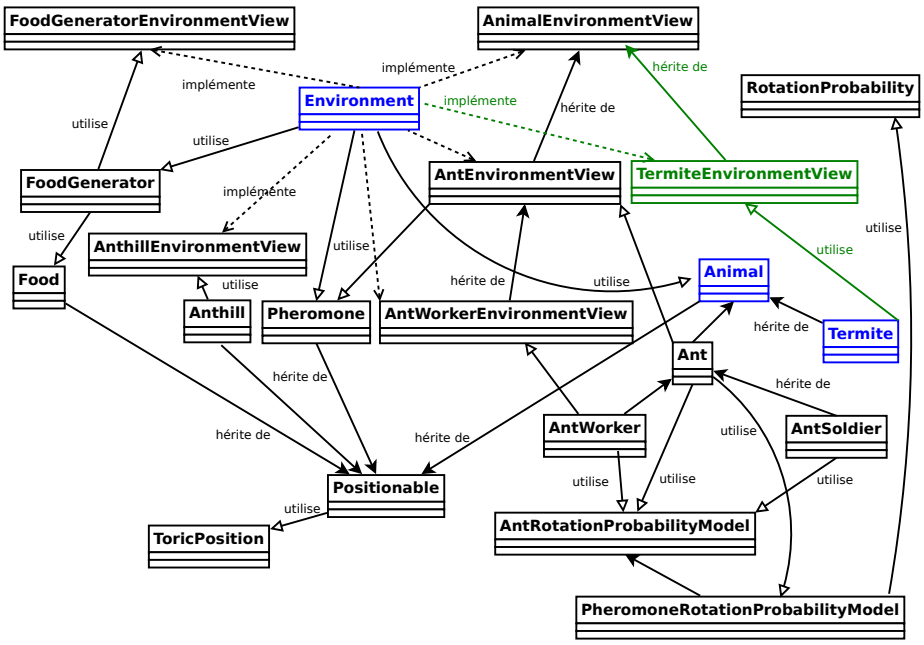

Figure 1: Composants impliqués dans l'étape 13 (en noir les composants déjà codés, en vert les nouveaux composants, en bleu ceux retouchés)

Nous allons dans cette étape compléter la modélisation des termites. Nous devrons pour cela, à l'image de ce que nous avons fait pour les fourmis, modéliser la vue qu'à une termite de l'environnement. Nous développerons ensuite un mécanisme de prédation et de combat qui permettra aux termites et aux fourmis soldates de jouer leur rôle spécifique dans la simulation. La mise en place de ce mécanisme nécessite de programmer des fonctionnalités de base permettant à chaque animal de savoir qui est un ennemi pour lui. Nous allons également, en préparation de la prochaine étape, mettre en place la notion de force d'attaque et de durée d'attaque.

La mise en place de ces nouveautés implique les composants illustrés dans la figure 1.

Implémentation

Termites

Les termites sont des animaux de l'environnement se déplaçant de façon aléatoire, comme nous les avons déjà vu le faire dans les premières étapes du projet.

Ce comportement n'était plus visible car nous n'avons pas encore mis en place les éléments de conception permettant de modéliser le fait qu'il s'agit bien là d'un comportement spécifique aux termites. C'est ce à quoi nous allons remédier maintenant.

En premier lieu, il s'agit de modéliser le fait qu'une termite n'a pas tout les droits possibles sur l'environnement et qu'elle ne peut agir que sur une vue particulière de ce dernier.

Pour cela créez l'interface `TermiteEnvironmentView` qui héritera de `AnimalEnvironmentView`. Dans le cadre de ce projet, cette vue ne contiendra rien de plus que ce qui est hérité `AnimalEnvironmentView` (elle sera donc vide), mais libre à vous de la modifier une fois le projet terminé ! Faites également en sorte que `Environment` implémente cette nouvelle interface.

Il s'agit ensuite de compléter dans notre conception, les éléments spécifiques aux termites; à l'image de ce nous avons déjà fait pour les fourmis. Les termites auront un rôle très proche des fourmis soldates. Notamment celui de se déplacer en quête de confrontation (méthode `seekForEnemies()`).

Voici donc les extensions à apporter au code.

- Dans `Termite` :
  - ajoutez `void seekForEnemies(AnimalEnvironmentView env, Time dt)` ; elle se contentera d'appeler `move` pour le moment ;
  - définissez `void specificBehaviorDispatch(AnimalEnvironmentView env, Time dt)` héritée de `Animal`, elle devra appeler `void`

`selectSpecificBehaviorDispatch(Termite termite, Time dt)` de `AnimalEnvironmentView` (voir plus bas) ;
  - définissez `void afterMoveDispatch(AnimalEnvironmentView env, Time dt)` héritée de `Animal`, elle devra appeler `void selectAfterMoveDispatch(Termite termite, Time dt)` de `AnimalEnvironmentView` (voir plus bas) ;
  - définissez `RotationProbability computeRotationProbsDispatch(AnimalEnvironmentView env)` héritée de `Animal`, elle devra appeler `void selectComputeRotationProbsDispatch(Termite termite, Time dt)` de `AnimalEnvironmentView` (voir plus bas) ;
  - ajoutez `RotationProbability computeRotationProbs(TermiteEnvironmentView env)` qui retournera simplement les probabilités associées aux angles par défaut (en appelant `computeDefaultRotationProbs` de `Animal`) ;
  - ajoutez `void afterMoveTermite(TermiteEnvironmentView env, Time dt)` ; cette méthode ne fera rien ;
- Dans `AnimalEnvironmentView` (et également dans `Environment`) :
  - déclarez (dans `AnimalEnvironmentView`) et définissez (dans `Environment`) `void selectSpecificBehaviorDispatch(Termite termite, Time dt)` ; les termites ayant un rôle analogue aux soldates, cette méthode appellera aussi la méthode `seekForEnemies()` ;
  - déclarez et définissez `void selectAfterMoveDispatch(Termite termite, Time dt)` qui appellera `afterMoveTermite` de `Termite` ;
  - déclarez et définissez `RotationProbability selectComputeRotationProbsDispatch(Termite termite)` qui appellera `computeRotationProbs` de `Termite` ;

Qui est l'ennemi ?

Comme les animaux ne seront pas tous égaux face à la prédation, il semble utile de programmer pour ces derniers une méthode telle que :

```
boolean isEnemy(Animal other)
```

retournant `true` si `other` est ennemi de `this`.

En effet, lors de ses déplacements un animal va être amené à en rencontrer d'autres et il devra naturellement se poser la question : « est-ce un ennemi pour moi ? », d'où le besoin de cette méthode `isEnemy()`.

Notes :

- évidemment, on ne sait pas définir la méthode `isEnemy()` au niveau d'abstraction des animaux ;
- une fourmi considérera comme ennemi une termite ou une fourmi d'une autre colonie, et une termite considérera comme ennemi toute fourmi.

[**fin de note**]

Imaginons maintenant comment on pourrait redéfinir concrètement la méthode `isEnemy()` pour des termites par exemple.

Dans la classe `Termite`, l'idée la plus évidente serait d'écrire :

```
@Override
boolean isEnemy(Animal other) {
  // si other est de type Termite alors retourner faux
  // si other est de type Ant, alors retourner true
}
```

Hum... à nouveau la tentation de faire de vilains tests de type !

Vous vous en doutez sûrement, un moyen de contourner cela est de recourir à nouveau à la technique du « *double-dispatch* ». L'idée est la suivante : on déclare dans `Animal` les méthodes abstraites :

```
abstract boolean isEnemy(Animal entity) ;
abstract boolean isEnemyDispatch(Termite other) ;
abstract boolean isEnemyDispatch(Ant other) ;
```

et la méthode `isEnemy()` dans `Termite` s'écrirait comme :

```
return !this.isDead() && !animal.isDead() && animal.isEnemyDispatch(this) ;`
```

(la termite a pour ennemi l'animal reçu, s'ils sont tous les deux vivants et que l'animal en question a pour ennemi la termite).

Avec, bien sûr, pour les termites `isEnemyDispatch(Termite termite)` retournant toujours faux (les termites ne sont pas ennemies entre elles) et `isEnemyDispatch(Ant ant)` retournant toujours vrai (toute termite est ennemie de n'importe quelle fourmi).

On procéderait de façon analogue pour `Ant`, avec simplement une méthode `isEnemyDispatch(Ant ant)` à peine plus compliquée.

Liste des ennemis visibles

Lors de la prochaine étape, où nous mettrons en œuvre les combats entre animaux, nous aurons besoin d'obtenir la liste des ennemis visibles pour un animal donné. Définissez pour cela dans `AnimalEnvironmentView` et implémentez dans `Environment` la méthode

- `List<Animal> getVisibleEnemiesForAnimal(Animal from)`

qui renvoie la liste des animaux qui sont des ennemies de `from` et qui se situe à une distance inférieure ou égale à `ANIMAL_SIGHT_DISTANCE`.

Nous aurons également besoin de savoir si un animal est visible ou non par des ennemis. Définissez dans `AnimalEnvironmentView` (et implémentez dans `Environment` la méthode

- `boolean isVisibleFromEnemies(Animal from)`

qui renvoie `true` si l'animal `from` est visible par des ennemis qui se situe à une distance inférieure ou égale à `ANIMAL_SIGHT_DISTANCE`.

Les méthodes `isVisibleFromEnemies(Animal from)` et `getVisibleEnemiesForAnimal()` ont des portions de traitements très ressemblantes. Pensez à bien modulariser leur implémentation pour éviter la duplication de code.

Dégâts & durée d'attaque

Nous allons pour finir ajouter quelques propriétés supplémentaires aux animaux qui seront utiles pour la mise en œuvre des combats lors de la prochaine étape. Il s'agit de la force d'attaque minimale et maximale, et de la durée de l'attaque.

La force d'attaque représente, comme son nom le suggère, les dégâts qu'inflige un animal à son adversaire. Nous aimerions que les dégâts infligés ne soient pas toujours les mêmes, ceux-ci seront bornés entre la force d'attaque minimale et la force d'attaque maximale. A chaque coup, l'animal infligera ainsi des dégâts de manière aléatoire entre sa force d'attaque minimale et maximale. Nous souhaiterions également que les animaux arrêtent le combat au bout d'un certain temps (sans qu'il y ait de vainqueur à proprement parler). Ce temps sera modélisé par la durée de l'attaque : une fois ce laps de temps écoulé, les animaux quitteront le combat et s'enfuiront.

Comme pour la vitesse, nous aimerions que chacune de ces propriétés soient spécifiques aux sous-classes et qu'elles puissent facilement être changées à partir d'un fichier de configuration. Nous allons donc employer la même technique, à savoir l'utilisation de méthodes virtuelles. Vous définirez les méthodes abstraites suivantes dans `Animal` :

- `int getMinAttackStrength()` qui retourne la force d'attaque minimale de l'animal ;
- `int getMaxAttackStrength()` qui retourne la force d'attaque maximale de l'animal ;
- `Time getMaxAttackDuration()` qui retourne la durée de l'attaque de l'animal avant que celui-ci ne quitte le combat.

Ces méthodes devront renvoyer les valeurs respectivement associées aux paramètres `ANT_WORKER_MIN_STRENGTH`, `ANT_WORKER_MAX_STRENGTH` et `ANT_WORKER_ATTACK_DURATION` pour `AntWorker`, et respectivement `ANT_SOLDIER_MIN_STRENGTH`, `ANT_SOLDIER_MAX_STRENGTH` et `ANT_SOLDIER_ATTACK_DURATION` pour `AntSoldier`. En ce qui concerne `Termite`, elles retourneront `TERMITE_MIN_STRENGTH`, `TERMITE_MAX_STRENGTH` et `TERMITE_ATTACK_DURATION`, respectivement.

Comme nous avons besoin de compter le temps qui s'est écoulé depuis le début du combat, nous allons déclarer (dans `Animal`) un champ de type `Time` nommé `attackDuration`. Il sera initialisé à `Time.ZERO` dans le constructeur.

Note : n'oubliez pas les inclusions statiques nécessaires à l'utilisation des constantes.

Tests et soumission

Tests locaux

Nous vous encourageons, comme toujours à ajouter vos propres tests ponctuels dans le fichier `ch/epfl/moocprog/tests/Main.java`.

Voici un exemple de code dont vous pouvez vous inspirer pour compléter ce fichier et tester les éléments ajoutés à votre programme lors de cette étape :

```
// quelques tests pour l'étape 13
System.out.println("\nA termite is added to an empty environment:");
ToricPosition termitePosition = new ToricPosition(20, 30);
Termite termit = new Termite(termitePosition);
ToricPosition positionBeforeUpdate = termitePosition;
System.out.println("Characteristics of the termite:");
System.out.println(termit);

env = new Environment();
env.addAnimal(termit);
env.update(Time.fromSeconds(1.));

// on teste si la termite est à nouveau capable de se déplacer
boolean hasMoved = !positionBeforeUpdate.equals(termit.getPosition());
System.out.print("The termite is now able to move : " );
System.out.println(hasMoved + "\n");

// on vérifie les probabilités de rotation
System.out.println("The rotation probabilities for the termite are:");
RotationProbability rotProbs = env.selectComputeRotationProbsDispatch(termit);
System.out.println("Angles :" + Arrays.toString(rotProbs.getAngles()));
System.out.println("Probabilities : " + Arrays.toString(rotProbs.getProbabilities()));
System.out.println();
// On vérifie les tests sur les ennemis
Termite termit2 = new Termite(termitePosition);
env.addAnimal(termit2);
System.out.print("Is a termite the ennemy of another termite : ");
System.out.println(termit2.isEnemy(termit));
```

```
Anthill anthill1 = new Anthill(new ToricPosition(10, 20));
AntWorker worker1 = new AntWorker(new ToricPosition(22, 28), anthill.getAnthillId());
env.addAnimal(worker1);
System.out.print("Is a termite the ennemy of a worker ant : ");
System.out.println(termit2.isEnemy(worker1));
// faire la même choses pour les autres combinaisons possibles
// (soldates, fourmis d'une même foumilières etc.)

// On vérifie les méthodes permettant la détection d'ennemis:
System.out.print("Can the worker ant be seen by an ennemy :");
System.out.println(env.isVisibleFromEnemies(worker1));
System.out.println("Characteristics of the visible ennemies :");
System.out.println(env.getVisibleEnemiesForAnimal(worker1));
// à compléter avec d'autres configuration pour les distances

// On vérifie les forces et temps d'attaque
 System.out.println((termit2.getMinAttackStrength()
      == getConfig().getInt(TERMITE_MIN_STRENGTH)));

 System.out.println((termit2.getMaxAttackStrength()
      == getConfig().getInt(TERMITE_MAX_STRENGTH)));

 System.out.println(termit2.getMaxAttackDuration().equals(
         getConfig().getTime(TERMITE_ATTACK_DURATION)));
```

Ceci devrait produire l'affichage suivant :

```
A termite is added to an empty environment:
Characteristics of the termite:
Position : 20.0, 30.0
Speed : 120.0
HitPoints : 500
LifeSpan : 30.000000

The termite is now able to move : true

The rotation probabilities for the termite are:
Angles :[-3.141592653589793, -1.7453292519943295, -0.9599310885968813,
-0.4363323129985824, -0.17453292519943295, 0.0, 0.17453292519943295,
0.4363323129985824, 0.9599310885968813, 1.7453292519943295, 3.141592653589793]
Probabilities : [0.0, 0.0, 5.0E-4, 0.001, 0.005, 0.987, 0.005, 0.001, 5.0E-4, 0.0, 0.0]

Is a termite the ennemy of another termite : false
Is a termite the ennemy of a worker ant : true
Can the worker ant be seen by an ennemy :true
Characteristics of the visible ennemies :
[Position : 20.0, 30.0
```

```
Speed : 120.0
HitPoints : 500
LifeSpan : 30.000000
]
true
true
true
```

Pour tester graphiquement le comportement global de votre programme, tout en préparant un peu le terrain aux tests de la prochaine étape, nous vous suggérons d'utiliser, en plus du fichier `config.cfg`, un fichier `config-step13.cfg` plus épuré contenant une fourmilière et une quarantaine de termites (toutes à la même position de départ, ce n'est pas très important). Cet environnement infesté de termites devrait vous permettre d'observer plus facilement les combats à la prochaine étape.

Vous devriez pouvoir vérifier graphiquement :

- que tous les comportements antérieurement codés pour les fourmis sont préservés ;
- que les termites ont retrouvé leur capacité à se mouvoir ;
- que les termites sont encore indifférentes à la présence de fourmis (ce à quoi nous allons remédier à la prochaine étape).

Une petite vidéo d'exemple d'exécution au terme de cette étape est disponible dans le matériel de la semaine 7.

Soumission

Pour soumettre votre devoir au correcteur automatique, il faut créer un fichier ZIP contenant **tous** vos fichiers sources personnels, depuis la racine `ch/` ; ceux de cette étape, mais aussi ceux de l'étape précédente. Concrètement, pour ce devoir ci, votre fichier ZIP doit donc contenir exactement les fichiers suivants :

```
ch/epfl/moocprog/AnimalEnvironmentView.java
ch/epfl/moocprog/Animal.java
ch/epfl/moocprog/AntEnvironmentView.java
ch/epfl/moocprog/AnthillEnvironmentView.java
ch/epfl/moocprog/Anthill.java
ch/epfl/moocprog/Ant.java
ch/epfl/moocprog/AntRotationProbabilityModel.java
ch/epfl/moocprog/AntSoldier.java
ch/epfl/moocprog/AntWorkerEnvironmentView.java
ch/epfl/moocprog/AntWorker.java
ch/epfl/moocprog/Environment.java
ch/epfl/moocprog/FoodGeneratorEnvironmentView.java
```

`ch/epfl/moocprog/FoodGenerator.java`
`ch/epfl/moocprog/Food.java`
`ch/epfl/moocprog/Pheromone.java`
`ch/epfl/moocprog/PheromoneRotationProbabilityModel.java`
`ch/epfl/moocprog/Positionable.java`
`ch/epfl/moocprog/RotationProbability.java`
`ch/epfl/moocprog/TermiteEnvironmentView.java`
`ch/epfl/moocprog/Termite.java`
`ch/epfl/moocprog/ToricPosition.java`

Note : si c'est plus pratique pour vous, vous *pouvez* aussi faire un zip qui contient tout le sous-répertoire `ch/epfl/moocprog`, y compris, donc, les fichiers que nous vous avons fournis. Ces fichiers seront simplement ignorés par le correcteur automatique (et remplacés par les nôtres). **Avant de passer à l'étape suivante, n'oubliez pas de faire une sauvegarde de l'étape en cours**, comme indiqué dans le même document.